

REPAST STATECHARTS GUIDE

JONATHAN OZIK, NICK COLLIER - REPAST DEVELOPMENT TEAM

0. BEFORE WE GET STARTED

Before we can do anything with Repast Symphony, we need to make sure that we have a proper installation of Repast Symphony 2.1. Instructions on downloading and installing Repast Symphony on various platforms can be found on the Repast website.

1. GETTING STARTED WITH STATECHARTS

Agent states and transitions between states are an important abstraction in agent-based modeling. While it is possible for Repast Symphony users to create their own implementation of state-based agent behaviors (e.g., by adapting the State pattern in Gamma et al. 1994¹) and even agent state visualizations, the effort involved in doing so is usually prohibitive. By integrating an agent statecharts framework into Repast Symphony, we made it easy for users of all levels to take advantage of this important modeling paradigm. Statecharts are visual representations of states and the transitions between those states². Statecharts can be very effective in visually capturing the logic within agents and quickly conveying the underlying dynamics of complex models.

Figure 1 shows a simple example of a statechart created with the Repast Symphony statecharts framework. The logic embedded in the diagram is mapped directly to the execution logic of an agent-based model. The benefits of the Repast Symphony statecharts framework include: improved clarity of a model's logic for model design, improved turnaround times for developing complex state based agent models, and the ability to convey in a compelling manner the internal state of agents as a simulation evolves to both experienced agent-modelers and to non-modelers alike. In the rest of this guide we will present the Repast Symphony statecharts framework.

1.1. Adding Statecharts. A statechart can be added to any Java, Groovy or ReLogo class. Right-clicking the class of interest and selecting New → Statechart Diagram (Figure 2) will bring up the new statechart wizard (Figure 3). The editable new statechart wizard elements are:

Date: July 1, 2013.

¹Gamma, E., R. Helm, R. Johnson, and J. M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. illustrated ed. Addison-Wesley Professional, 1994.

²Statecharts were first proposed by Harel in Harel, D., 1987. Statecharts: A visual formalism for complex systems. Sci. Comput. Program., 8(3), pp.231274.

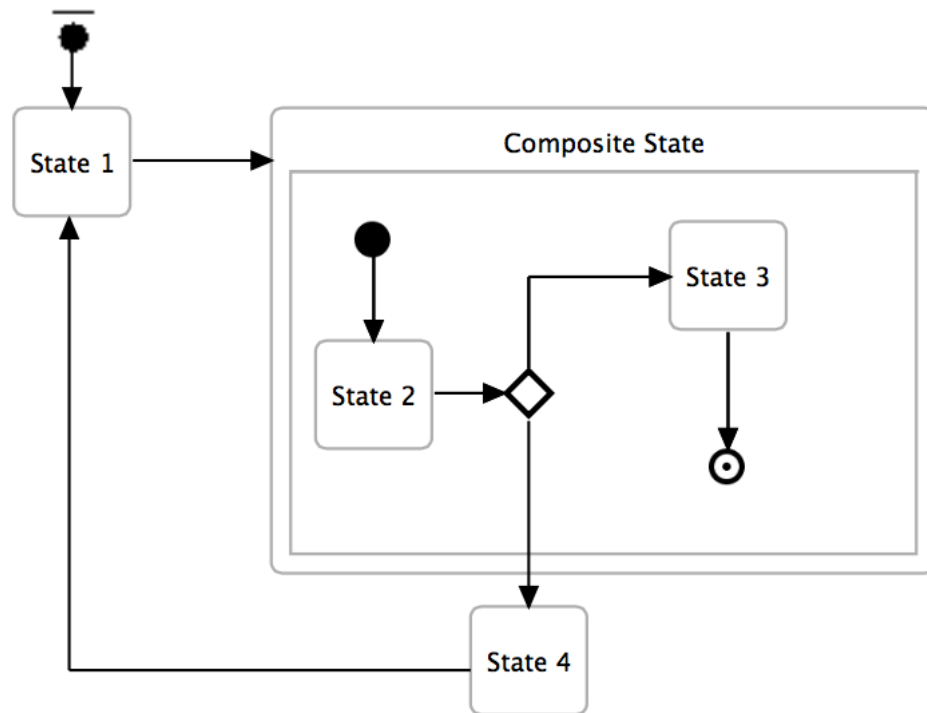


FIGURE 1. An example statechart created with the Repast Symphony visual statecharts editor.

File Name: This is the .rsc statechart file name that will be edited with the visual statecharts editor.

Name: The display name of the statechart.

Class Name: The name of the statechart class which will be generated.

Package: The package name within the **src-gen** source folder where the statechart class source code will be generated.

Agent Class: This is the agent class that will be associated with the statechart.

After accepting or modifying the defaults, clicking the *Finish* button will bring up the newly created blank statechart editor.

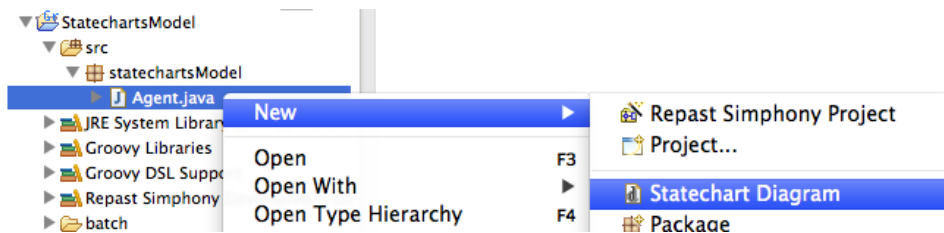


FIGURE 2. Creating a new statechart.

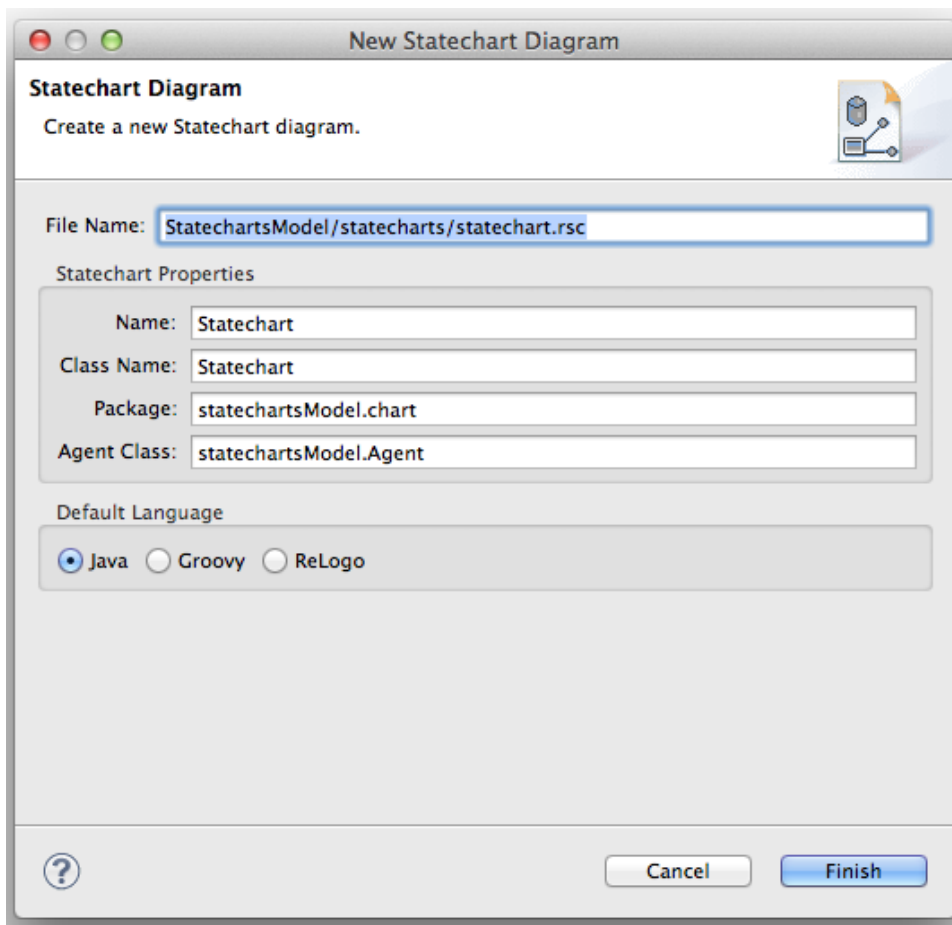


FIGURE 3. New statechart wizard.

1.2. Statecharts Editor. The statecharts editor is divided into three main panels (Figure 4). The first (Figure 4a) is the statecharts workspace area. This is where the visual elements of a statechart are created and arranged. The palette panel (Figure 4b) shows the available statechart elements that can be used in the statecharts workspace³ Clicking on an element in the palette panel and then clicking on the statecharts workspace will create an instance of that element in the workspace. The properties panel (Figure 4c) shows the properties of the element selected in the statecharts workspace. If, as in Figure 4, no element is selected, the properties of the statechart itself are shown. In addition to displaying element properties, the panel is also where the properties of elements can be edited. A statechart has a priority which indicates the order in which it will be resolved with respect to other statecharts (see red box in Figure 5). So, for example, if an agent has two statecharts (A and B) and statechart A should be resolved before statechart B, giving statechart A a higher priority will ensure that this occurs.

There is a contextual menu approach for adding elements to the workspace as well. Simply hovering over an area in the workspace will reveal a contextual menu of the available elements appropriate for the region pointed to. If the mouse pointer is on an empty space in the workspace background, you will see the contextual menu in Figure 6. If the pointer is on a state, you will see transitions shortcuts like in Figure 7, the left symbol indicating a connection from this state and the right symbol a connection to this state. Finally, if the pointer is inside a composite state, you will see the contextual menu in Figure 8.

³Sections 2 and 3 will cover these elements in detail.

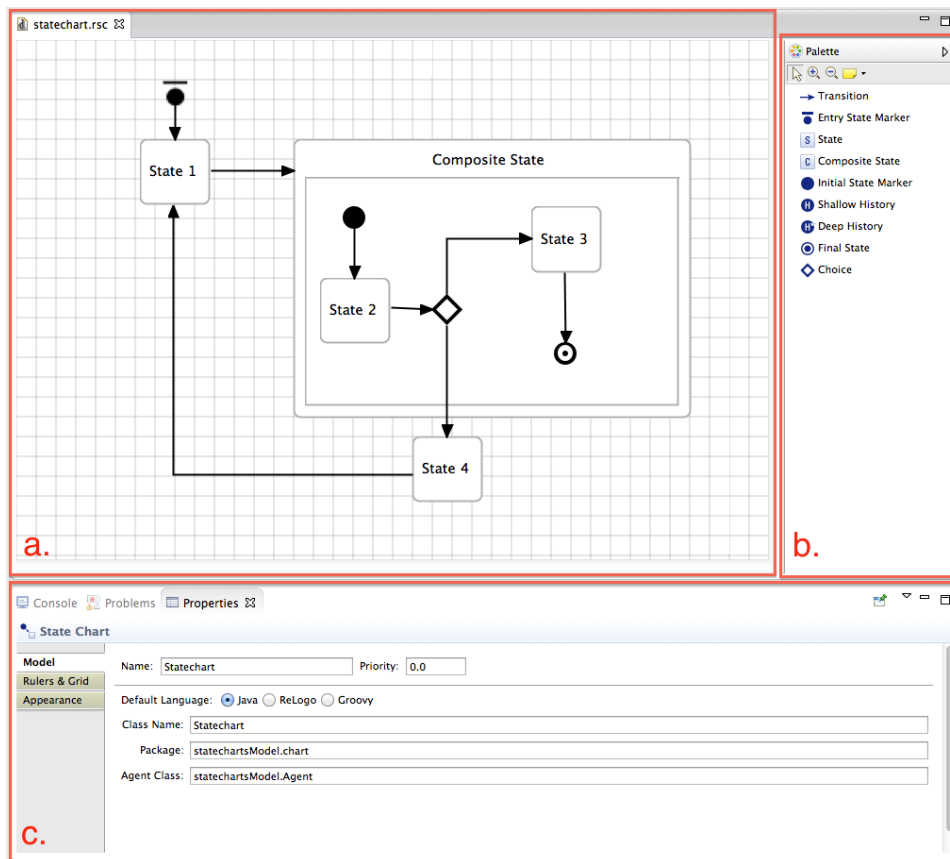


FIGURE 4. The components of the statecharts visual editor. a) The workspace area where the statechart elements are created and arranged. b) The palette panel of available elements, including selection, zoom and note tools. c) The properties panel of the selected statechart element in the workspace (a). If no element is selected, the properties of the statechart itself are displayed.

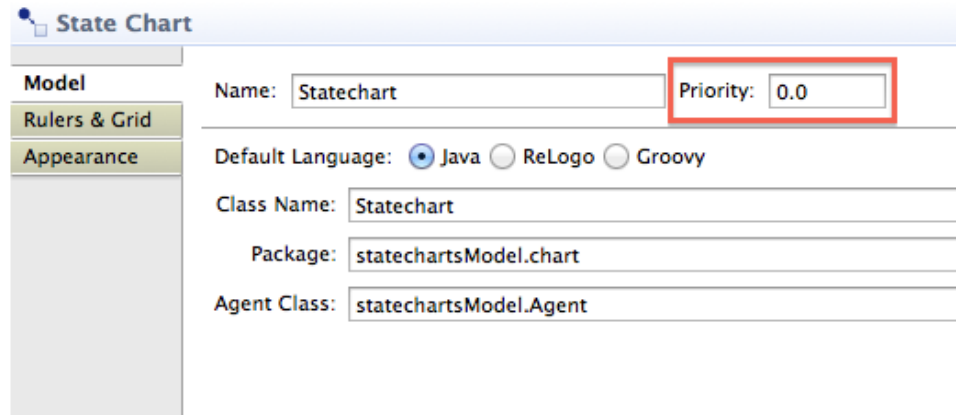


FIGURE 5. The statechart properties panel with the priority element indicated by a red box.

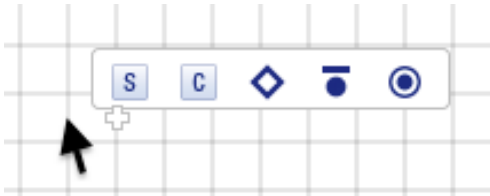


FIGURE 6. The default contextual menu when the pointer is on a blank area of the workspace.

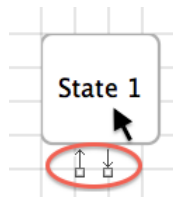


FIGURE 7. The transitions shortcuts, circled in red, for making connections from (left) and to (right) the state.

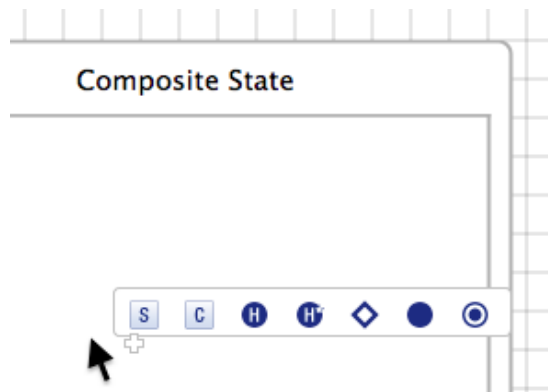


FIGURE 8. The contextual menu when the pointer is inside a composite state.

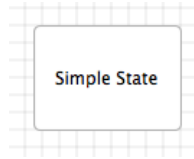


FIGURE 9. Simple state.

2. STATES

One of the fundamental building blocks of statecharts are states. Here we introduce the different types of states that exist within the Repast Symphony statecharts framework.

2.1. Entry State Marker.

Every statechart must have an entry state marker. This defines point of entry into a statechart takes when the statechart is activated.

2.2. Simple State.

A simple state looks like Figure 9. At any one point in time within an active statechart, one and only one of the simple states will be active. In addition to their *ID*, simple states can have *On Enter* and *On Exit* actions defined, as seen in the simple state properties panel in Figure 10. These actions are triggered when entering or exiting the simple state, respectively. The keywords available within the two action blocks are:

agent: This is the agent that contains the statechart. Any method (e.g., `customMethod`) defined on the agent can be invoked through this reference (e.g., `agent.customMethod()`).

state: This is the state itself. For example, the state's *ID* can be accessed via `state.getId()`.

params: This is the model's `Parameters` object. As an example, a double valued parameter `dParam` can be retrieved with: `params.getDouble("dParam")`⁴.

As is the case with all types of action blocks, their logic can be specified using Java, Groovy or ReLogo. Specifically, any Java, Groovy or ReLogo code can be used to express the behavior that should be executed upon entry to or exit from the state⁵.

⁴See the source or JavaDoc for `repast.simphony.parameter.Parameters` for all of the available methods.

⁵When using the ReLogo option, the `agent` parameter is implicit so writing `customMethod()` is equivalent to `agent.customMethod()`.

S State

Model

Appearance

ID: Simple State

Language: ☒ Java ☐ ReLogo ☐ Groovy

On Enter:

On Exit:

FIGURE 10. Simple state properties.

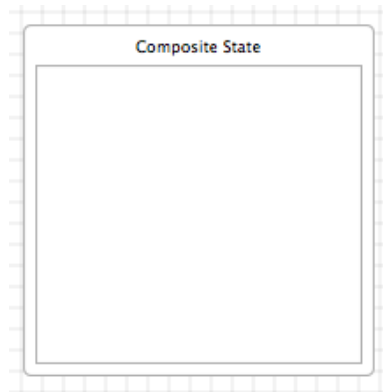


FIGURE 11. Composite state.

2.3. Composite State.

Composite states are used to nest elements within a statechart. Figure 11 shows an empty composite state and Figure 12 shows the properties panel for composite states, which is identical to that of the simple states in that *On Enter* and *On Exit* actions can be defined. The difference between composite and simple states lies in the fact that composite states can include the following elements as sub-elements:

- Simple state (Section 2.2)
- Composite state (Section 2.3)
- Initial state marker (Section 2.4)
- History state (Section 2.5)
- Final state (Section 2.6)
- Branching state (Section 2.7)

Whenever a sub-element is active, the composite state containing that sub-element will be active as well. If a transition is made from outside of a composite state directly to a sub-element, the composite state will be entered prior to its sub-elements. In a similar manner, if a transition is followed from a sub-element out of the composite state, the composite state will be exited after the sub-elements are exited.

Composite State

Model

Appearance

ID: Composite State

Language: ☒ Java ☐ ReLogo ☐ Groovy

On Enter:

On Exit:

FIGURE 12. Composite state properties.

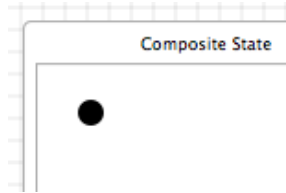


FIGURE 13. Initial state marker (within a composite state).

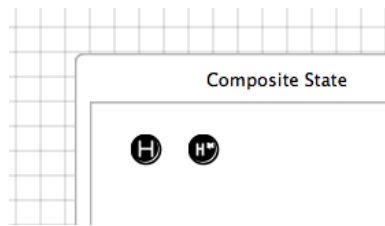


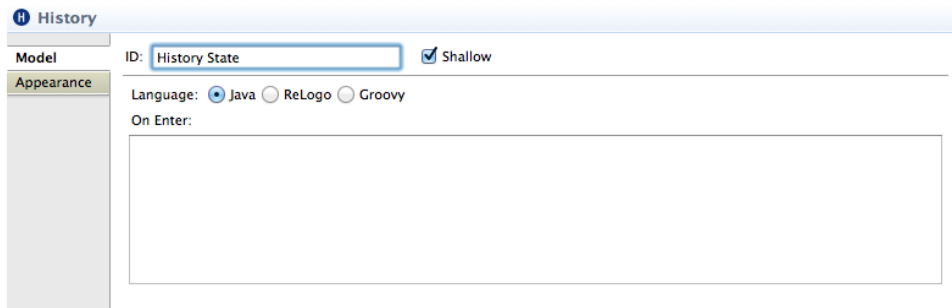
FIGURE 14. Shallow (left) and deep (right) history states (within a composite state).

2.4. Initial State Marker. ●

Any composite state that has a transition ending at it or contains history states must define an initial state marker. The initial state marker points to the element within the composite state that should be entered upon entering the composite state.

2.5. History State. H H*

There are two types of history states, shallow and deep (Figure 14). When a shallow history state is entered, the last active element within the enclosing composite state at the same hierarchical level of the history state is re-entered. For a deep history state, the last active *simple state* within the enclosing composite state, no matter at what level of the nesting hierarchy, is entered. In both cases if there was no previously active state, the state pointed to by the initial state marker is entered. Figure 15 shows the properties panel for a (shallow) history state. Only an *On Enter* element can be defined since history states are never directly exited.




The screenshot shows the 'History' properties panel in the REPAST Statecharts IDE. The panel has a sidebar on the left with 'Model' and 'Appearance' tabs. The 'Model' tab is active, showing the following fields:

- ID:** A text box containing 'History State'.
- Shallow:** A checked checkbox.
- Language:** Radio buttons for 'Java' (selected), 'ReLogo', and 'Groovy'.
- On Enter:** A large empty text area for defining actions.

FIGURE 15. The properties panel for a shallow history state. A deep history state would have the same properties panel except that the *Shallow* element would be unchecked.

The image shows a software interface for configuring a state. At the top, a blue header bar contains a radio button icon and the text "Final State". Below this, on the left, is a vertical sidebar with two tabs: "Model" and "Appearance". The "Model" tab is currently selected. To the right of the sidebar, under the "Model" tab, there is an "ID:" label followed by a text input field containing the text "Final State". Below the ID field, there is a "Language:" label followed by three radio buttons: "Java" (which is selected), "ReLogo", and "Groovy". Below the language selection, there is an "On Enter:" label followed by a large, empty rectangular text area for defining actions.

FIGURE 16. The properties panel for a final state.

2.6. Final State.  A final state marks the end of all activities for a statechart. When a final state is entered, no further states will be visited and no transitions will be triggered. Figure 16 shows the properties panel for a final state. Only an *On Enter* element can be defined since final states are never exited.

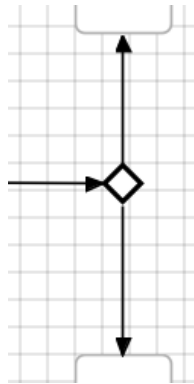


FIGURE 17. A branching state with one incoming and two outgoing transitions.

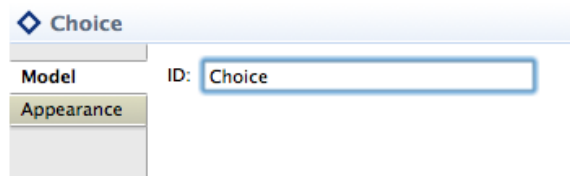


FIGURE 18. The properties panel for a branching state.

2.7. Branching State.

Branching states represent logical branching within statecharts (Figure 17). Every branching state must define one outgoing *Default* transtion, where the rest of the outgoing transitions are *Condition* transitions (Section 3.4). The *Condition* transitions are checked for validity and, if valid conditions are found, the transitions' priorities dictate the transition that is followed. If no valid transitions are found, the *Default* transition is followed. Figure 18 shows the properties panel for a branching state. Since a branching state is entered and immediately exited, nothing other than the state's *ID* can be specified.

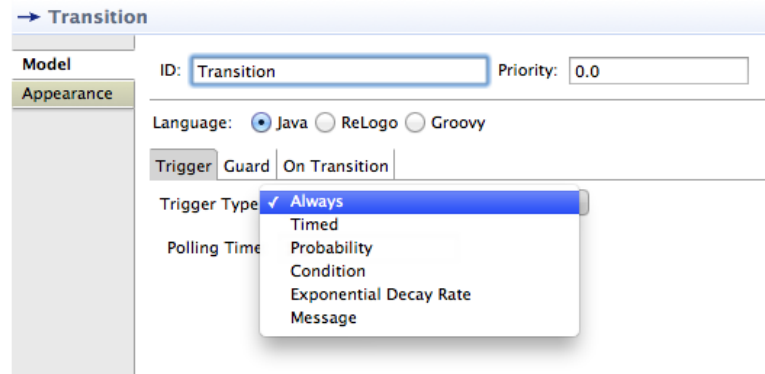


FIGURE 19. The properties panel showing the different types of transitions that are available.

3. TRANSITIONS

Transitions between states make up the other fundamental building block of statecharts. In this section we introduce the different types of transitions that can be used within the Repast Symphony statecharts framework.

There are two overall types of transitions, *regular transitions* (Figure 20) which connect different states and *self transitions* (Figure 21) which are internal to a state⁶. There are a number of different transition trigger types, demonstrated in the transition properties panel in Figure 19 (these will be discussed below in further detail).

For any transition an *On Transition* action can be defined (see Figure 22). This action will be executed whenever the transition is traversed. The keywords available within the *On Transition* action block are:

agent: This is the agent that contains the statechart.

transition: This is the transition itself. For example, the transition's source state can be accessed via: `transition.getSource()`⁷.

params: This is the model's `Parameters` object.

For almost all types of transitions⁸ a *Guard* condition can be defined (see Figure 23). A *Guard* condition is an additional boolean condition that has to be satisfied for a transition that is valid to be actually considered as a candidate for traversal. This condition is

⁶One also has the ability to define a *regular transition* that begins and ends at the same state. Unlike the *self transition* case, each time the *regular transition* is taken, the state will be exited and subsequently re-entered.

⁷See the source or JavaDoc for `repast.simphony.statecharts.Transition` for all of the available methods.

⁸All transitions except default transitions out of branching states.

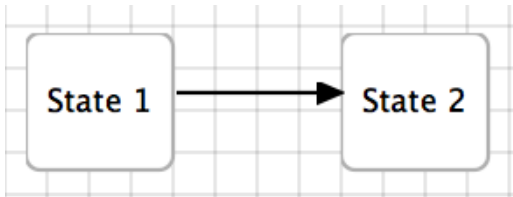


FIGURE 20. Regular transition between states 1 and 2.

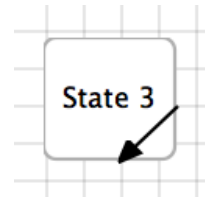


FIGURE 21. Self transition internal to State 3.

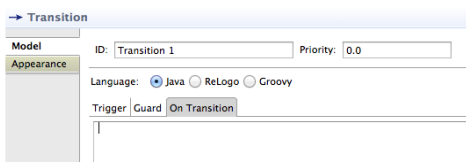


FIGURE 22. Properties panel for a transition showing the *On Transition* action block.

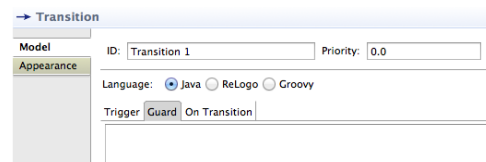


FIGURE 23. Properties panel for a transition showing the *Guard* condition block.

specified by a block of code that returns a boolean and the keywords available in a *Guard* condition are the same as those in an *On Transition* action block.

When there is more than one valid transition ties are broken using the priority of the transition. If the priorities of valid transitions are equal then one of the transitions will be chosen with a uniform random probability. The priority of a transition can be specified in the transition's properties panel.

Regular transitions can be divided into zero time transitions and non-zero time transitions⁹. For zero time transitions when a new state is entered, if there is a valid zero time transition out of it, that transition is followed immediately. Always (Section 3.1), Condition (Section 3.4) and Message (Section 3.6) transition triggers are zero-time transitions.

Every transition has a *polling time* associated with it. This indicates the frequency (in units of simulation *ticks*) with which the transition is polled for validity.

Next we present the different transition trigger types in more detail.

⁹All *self transitions* are non-zero time transitions.

The image shows a software interface titled "Transition" with a blue header bar. On the left, there is a sidebar with two tabs: "Model" and "Appearance", with "Appearance" currently selected. The main area contains the following fields:

- ID:** A text box containing "Always transition".
- Priority:** A text box containing "0.0".
- Language:** Three radio buttons labeled "Java", "ReLogo", and "Groovy". The "Java" button is selected.
- Trigger:** A tabbed interface with three tabs: "Trigger", "Guard", and "On Transition". The "Trigger" tab is active.
- Trigger Type:** A dropdown menu showing "Always".
- Polling Time:** A text box containing "1.0".

FIGURE 24. The properties panel for an *always* transition.

3.1. Always Trigger. *Always* triggers are always valid. The transition can, however, contain a *Guard* condition which if false, would prevent the transition from triggering. Because this trigger type results in zero time transitions, it is important to make sure that there are no *always* trigger transitions contributing to zero time loops in any statechart you create, since this has the potential to create never-ending loops. The properties panel for an *always* trigger transition is in Figure 24. One use for *always* trigger transitions is as self transitions to execute some action at a set polling time.

The screenshot shows a software interface for configuring a transition. At the top, a blue header bar contains a right-pointing arrow and the word "Transition". Below this, on the left, is a vertical sidebar with two tabs: "Model" and "Appearance". The "Model" tab is currently selected. The main area to the right of the sidebar contains the following fields:

- ID:** A text box containing "Timed Transition".
- Priority:** A text box containing "0.0".
- Language:** Three radio buttons labeled "Java", "ReLogo", and "Groovy". The "Java" radio button is selected.
- Trigger:** A tabbed interface with three tabs: "Trigger", "Guard", and "On Transition". The "Trigger" tab is selected.
- Trigger Type:** A dropdown menu with "Timed" selected.
- Time:** A large, empty text area for entering code.

FIGURE 25. The properties panel for a *timed* transition.

3.2. Timed Trigger. *Timed* triggers become valid after some time, measured in simulation *ticks*. The properties panel for a *timed* trigger transition is shown in Figure 25. The *Time* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., 2 or `agent.getDelay()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). If at the time a *timed* trigger is valid a *Guard* condition keeps the transition from being valid, the transition does not get reinitialized and will simply remain invalid.

The screenshot shows the 'Transition' properties panel in Repast. On the left, a sidebar contains 'Model' and 'Appearance' tabs. The main panel is titled 'Transition' and contains the following fields:

- ID:** A text box containing 'Probability Transition'.
- Priority:** A text box containing '0.0'.
- Language:** Three radio buttons: 'Java' (selected), 'ReLogo', and 'Groovy'.
- Trigger:** A tabbed interface with 'Trigger', 'Guard', and 'On Transition' tabs. The 'Trigger' tab is active.
- Trigger Type:** A dropdown menu showing 'Probability'.
- Polling Time:** A text box containing '1.0'.
- Probability:** A large, empty text area for entering code.

FIGURE 26. The properties panel for a *probability* transition.

3.3. Probability Trigger. *Probability* triggers are evaluated as valid with a specified probability. The properties panel for a *probability* trigger transition is shown in Figure 26. The *Probability* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., 0.2 or `agent.getProbability()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated each time the transition is polled for validity.

The screenshot shows the 'Transition' properties panel in the REPAST Statecharts GUI. The panel is titled '→ Transition' and has two tabs: 'Model' and 'Appearance'. The 'Model' tab is active. It contains the following fields and controls:

- ID:** A text field containing 'Condition Transition'.
- Priority:** A text field containing '0.0'.
- Language:** Three radio buttons labeled 'Java' (selected), 'ReLogo', and 'Groovy'.
- Trigger:** A tabbed interface with three tabs: 'Trigger' (selected), 'Guard', and 'On Transition'.
- Trigger Type:** A dropdown menu showing 'Condition'.
- Polling Time:** A text field containing '1.0'.
- Condition:** A large text area for entering the condition code.

FIGURE 27. The properties panel for a *condition* transition.

3.4. Condition Trigger. *Condition* triggers are evaluated as valid based on a specified condition. The properties panel for a *condition* trigger transition is shown in Figure 27. The *Condition* element in the properties panel accepts general Java, Groovy or ReLogo code returning a boolean value, including simple boolean entries (e.g., `true` or `agent.getCondition()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated each time the transition is polled for validity.

→ Transition

Model ID: Priority:

Appearance

Language: ☒ Java ☐ ReLogo ☐ Groovy

Trigger: ☒ Guard ☐ On Transition

Trigger Type:

Exponential Decay Rate:

FIGURE 28. The properties panel for a *exponential decay rate* transition.

3.5. Exponential Decay Rate Trigger. *Exponential decay rate* triggers become valid after a random time following the exponential distribution. The properties panel for an *exponential decay rate* trigger transition is shown in Figure 28. The *Exponential Decay Rate* element in the properties panel accepts general Java, Groovy or ReLogo code returning a numerical value, including simple numerical entries (e.g., 2 or `agent.getDecayRate()`), with the same keywords as the *On Transition* action block (i.e., `agent`, `transition`, `params`). The code block is evaluated when the transition is initialized (i.e., when a state is entered that has a possible *exponential decay rate* transition leading out of it). The code block supplies the λ parameter to the exponential distribution specified by the probability density function:

$$(1) \quad f(t) = \lambda e^{-\lambda t}$$

The expected value of an exponentially distributed random variable with parameter λ is $1/\lambda$. So given, for example, a λ of 2, the expected value for the time it would take for an *exponential decay rate* transition to trigger would be 0.5 in units of simulation *ticks*.

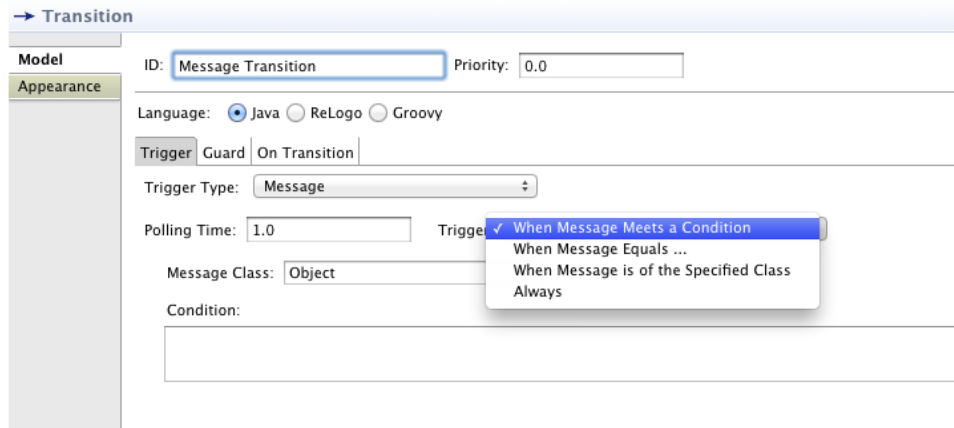


FIGURE 29. The properties panel for a *message* message, with the available trigger types shown under the *Trigger* element.

3.6. Message Trigger. *Message* triggers become valid when a message meeting specific criteria is received by the statechart. A statechart is sent a message when the statechart's `receiveMessage(Object)` method is called. From an agent-based modeling perspective, this would likely occur when an agent is sent a message and then the agent forwards the message to all or a subset of its statecharts¹⁰. There are four different types of *message* triggers, shown in the drop down menu of the *Trigger* element in the *message* trigger properties pane in Figure 29, and we present them next.

¹⁰There is nothing to prevent one agent from directly accessing another agent's statechart if the statechart is visible, but it could be considered not very good practice in an object oriented programming sense.

→ Transition

Model

Appearance

ID: Message Transition Priority: 0.0

Language: ☒ Java ☐ ReLogo ☐ Groovy

Trigger Guard On Transition

Trigger Type: Message

Polling Time: 1.0 Trigger: When Message Meets a Condition

Message Class:

Condition:

FIGURE 30. The properties panel for a *When Message Meets a Condition* message trigger.

3.6.1. *When Message Meets Condition*. The *When Message Meets a Condition* message trigger has the properties panel shown in Figure 30. The *Message Class* element specifies the type of the message, which can be any of the basic types in Figure 31 or the fully qualified name of any another type. To specify a type not in the list, enter the fully qualified name of the type in the combo box. The *Condition* element in the properties panel accepts general Java, Groovy or ReLogo code returning a boolean value, including simple boolean entries (e.g., `true` or `agent.getCondition()`). The keywords available within the *Condition* action block are:

agent: This is the agent that contains the statechart.

transition: This is the transition itself.

message: This is the message received by the statechart.

params: This is the model's `Parameters` object.

The code block is evaluated each time the transition is polled for validity.

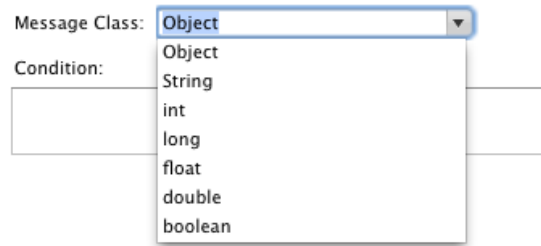


FIGURE 31. The basic types available for the *Message Class* element in the properties panel for the *When Message Meets a Condition*, *When Message Equals*, and *When Message is of the Specified Class* message triggers. Additional types can be specified with their fully qualified names.

The screenshot shows the 'Transition' properties panel in Repast. On the left, a sidebar has tabs for 'Model' and 'Appearance'. The main panel is titled 'Transition' and contains the following fields:

- ID:** A text box containing 'Message Transition'.
- Priority:** A text box containing '0.0'.
- Language:** Three radio buttons: 'Java' (selected), 'ReLogo', and 'Groovy'.
- Trigger:** A tabbed interface with 'Trigger', 'Guard', and 'On Transition' tabs. The 'Trigger' tab is active.
- Trigger Type:** A dropdown menu showing 'Message'.
- Polling Time:** A text box containing '1.0'.
- Trigger:** A dropdown menu showing 'When Message Equals ...'.
- Message Class:** A dropdown menu.
- Equals:** A large, empty text area for defining the equals condition.

FIGURE 32. The properties panel for a *When Message Equals* message trigger.

3.6.2. *When Message Equals*. The *When Message Equals* message trigger has the properties panel shown in Figure 32. The only difference between this and the *When Message Meets a Condition* message trigger is that instead of a *Condition* element there is an *Equals* element that needs to be defined. The *Equals* element accepts general Java, Groovy or ReLogo code returning any value that will be checked against the received message using the message's `equals(Object)` method. The keywords within the *Equals* block are the same as the *On Transition* action block (i.e., `agent`, `transition`, `params`).

The screenshot shows the 'Transition' properties panel in the REPAST Statecharts GUI. The 'Model' tab is selected, and the 'Appearance' sub-tab is active. The 'ID' field is set to 'Message Transition' and the 'Priority' is '0.0'. The 'Language' is set to 'Java'. The 'Trigger' tab is selected, and the 'Trigger Type' is 'Message'. The 'Polling Time' is '1.0'. The 'Trigger' dropdown is set to 'When Message is of the Specified Class'. The 'Message Class' dropdown is empty. The 'Equals' field is empty.

FIGURE 33. The properties panel for a *When Message is of Class* message trigger.

The screenshot shows the 'Transition' properties panel in the REPAST Statecharts GUI. The 'Model' tab is selected, and the 'Appearance' sub-tab is active. The 'ID' field is set to 'Message Transition' and the 'Priority' is '0.0'. The 'Language' is set to 'Java'. The 'Trigger' tab is selected, and the 'Trigger Type' is 'Message'. The 'Polling Time' is '1.0'. The 'Trigger' dropdown is set to 'Always'. The 'Message Class' dropdown is empty. The 'Equals' field is empty.

FIGURE 34. The properties panel for an *Always* message trigger.

3.6.3. *When Message is of Class*. The *When Message is of Class* message trigger has the properties panel shown in Figure 33. Any message that is received that is of the specified class type will result in the transition being triggered.

3.6.4. *Always*. The *Always* message trigger has the properties panel shown in Figure 34. Any message that is received will result in the transition being triggered.

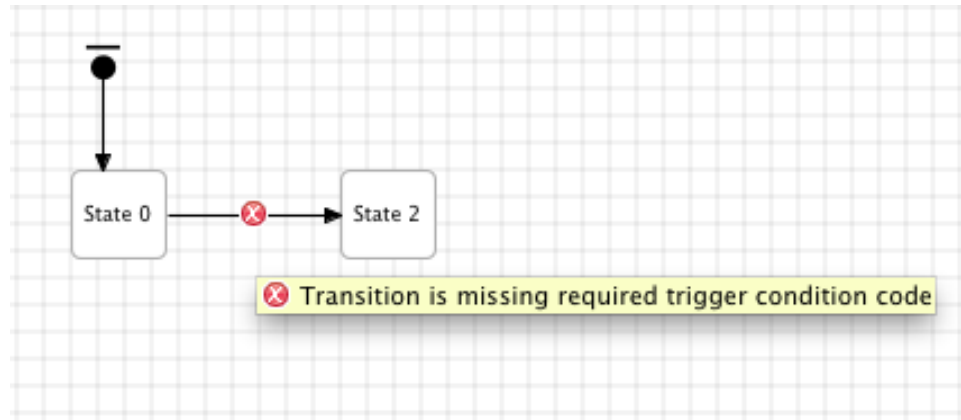


FIGURE 35. An error marker on a transition

4. DEBUGGING STATECHARTS

When a state chart is saved the code for that state chart will be generated in the `src-gen` directory in the state chart's project. At that time the state chart will also be validated and any structural warnings or errors will be displayed in the state chart workspace. Figure 35 is an example of this. The transition between State 0 and State 1 has an error. Moving the mouse pointer over the error marker will display a tooltip with the error message. In this case, the transition has a *Condition* trigger type but no condition has been specified in the transition's properties. The error message will also be displayed in Eclipse's Problems view.

If there is an error in the code generated by the state chart, you will see the error marker on the `src-gen` folder in the state chart's project. This kind of error will occur when any code specified in the state chart's state or transition properties (such as the *On Exit*, *Condition* and so forth code blocks) is erroneous. To fix these kind of errors, expand the `src-gen` folder to find the offending file as in figure 36. Open the file. The comments in the file describe the state chart element that produced the bad code and eclipse will flag errors in the code itself. Figure 37 shows such an error. The error is itself is that the `getHealt` method is not defined on the agent. The comments in the code state that this is the code for the "Condition trigger condition for Transition 3, from = State 0, to = State 2." To fix this then, we need to edit the trigger condition code in Transition 3. To find it, we can use the *to* and *from* states mentioned in the comments and select the transition that connects them. Note that editing the code directly in the `.java` file will remove the error, but will **NOT** fix the problem. The next time state chart is saved, the code will be regenerated. The code must be fixed in the state chart element that produced the problem.



FIGURE 36. An example of an error in the code generated from the state chart

```

/**
 * Condition trigger condition for Transition 3, from = State 0, to = State 2.
 */
@GeneratedFor("_cYcHg0JyEeKr6btr8V3khA")
public class SC2ConditionTriggerCondition1 implements
    ConditionTriggerCondition<Agent> {
    @Override
    public boolean condition(Agent agent, Transition<Agent> transition,
        Parameters params) throws Exception {
        return agent.getHealth() < 3;
    }
}

```

FIGURE 37. An error in the code produced by the state chart. Note that the comments refer to the element that produced the code.